Characterizing and Mitigating Work Time Inflation in Task Parallel Programs

Stephen L. Olivier
Department of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599–3175
olivier@cs.unc.edu

Bronis R. de Supinski and Martin Schulz
Lawrence Livermore National Laboratory
Livermore, CA 94550
{bronis, schulzm}@llnl.gov

Jan F. Prins
Department of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599–3175
prins@cs.unc.edu

***Abstract*—Task parallelism raises the level of abstraction in shared memory parallel programming to simplify the development of complex applications. However, task parallel applications can exhibit poor performance due to thread idleness, scheduling overheads, and *work time inflation* – additional time spent by threads in a multithreaded computation beyond the time required to perform the same work in a sequential computation. We identify the contributions of each factor to lost efficiency in various task parallel OpenMP applications and diagnose the causes of work time inflation in those applications.**

Increased data access latency can cause significant work time inflation in NUMA systems. Our locality framework for task parallel OpenMP programs mitigates this cause of work time inflation. Our extensions to the Qthreads library demonstrate that locality-aware scheduling can improve performance up to 3X compared to the Intel OpenMP task scheduler.

I. Introduction

Multicore computing has led to a renaissance for shared memory parallel programming models. The task parallel model shows particular promise due to its problem-centric expression of parallelism, including deeply nested and irregular computations. Many parallel programming languages and libraries allow programmers to express explicit tasks that are scheduled on available threads for execution, e.g., OpenMP 3.0 [1], Intel Threading Building Blocks [2], and Microsoft Task Parallel Library [3]. However, interactions between the application, the run time system, and the architecture complicate the performance and scalability of those applications [4], [5].

Recent work has made significant progress towards identifying and mitigating the causes of the observed performance gap. Tallent and Mellor-Crummey [6] divide the total execution time spent by all threads into three categories: *work time*, *idle time*, and *(parallelization) overhead time*. During work time, tasks perform useful computation, while idle time results from load imbalance and overhead time includes task creation, scheduling, and synchronization. They show that coarsening the granularity of tasks can decrease overhead time and, conversely, using finer-grained tasks can decrease idle time. However, load imbalance and overhead do not account for all observed performance loss. This paper explores another major cause, *work time inflation* – additional time spent by threads in a multithreaded computation beyond the time required to perform the same work sequentially – that can dominate performance loss in some applications.

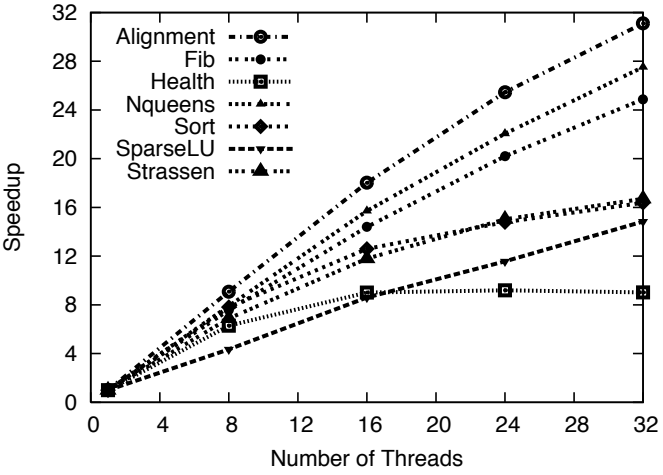


Fig. 1: BOTS speedups using the Intel OpenMP run time.

We make the following contributions:

- The characterization of lost efficiency in the execution of task parallel computations due to thread idleness, overhead costs in the scheduler, and *work time inflation*, along with the sources of work time inflation.
- A framework that enables locality-based task scheduling to minimize non-local memory accesses on NUMA architectures, featuring a concise mechanism for the programmer to specify the placement of tasks on locality domains, and a run time scheduler to support that mechanism.
- An empirical evaluation that demonstrates the effectiveness of our framework as implemented in extensions to the scheduler of an open-source OpenMP run time system on a modern multi-socket multicore NUMA architecture.

Our results demonstrate that locality-aware task scheduling significantly improves the performance of task parallel programming on NUMA systems: up to 2X over locality-oblivious scheduling within the same OpenMP implementation and up to a 3X over Intel's commercial implementation.

II. Diagnosing Sources of Lost Efficiency

Our diagnosis of lost efficiency focuses on the differences between sequential and multithreaded computations.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

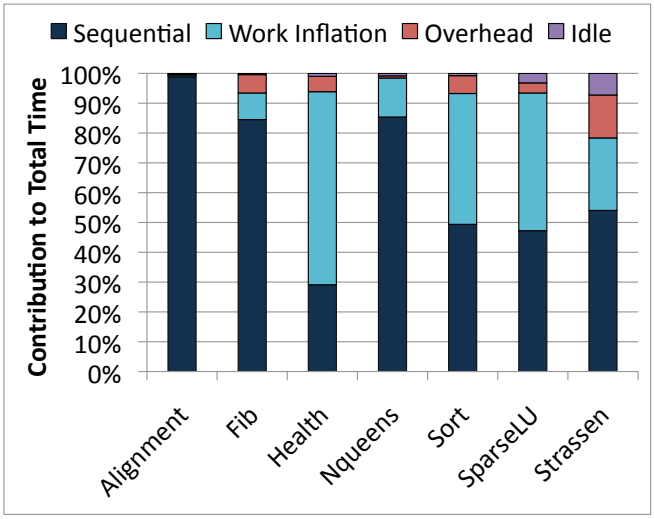


Fig. 2: Intel OpenMP 32-thread execution time breakdown.

Sequential computation uses a single thread that is never idle. Multithreaded execution can have idle threads at any particular time. The idle time occurs when no tasks are available to execute, for example due to load imbalance or dependencies between tasks. Sequential execution comprises a flow of execution without the distinct tasks that characterize task parallel multithreaded execution. The run time system manages activities that contribute to scheduling overhead: task creation, scheduling, synchronization, and retirement.

Apart from idle time and overhead, work time, which is the time to complete instructions of the computational work, accounts for the remaining time. Sequential equivalent time is the execution work time of a sequential equivalent of a task parallel program. The work time in a multithreaded execution often exceeds the sequential equivalent time. This *work time inflation* can have various causes, as discussed in the next section. Alternatively, negative work time inflation can also occur, for example, due to caching effects.

We use HPCToolkit [7] to measure the time spent by all threads in executions of task parallel programs from the Barcelona OpenMP Tasks Suite (BOTS) [8]:

- *Alignment*: Uses dynamic programming to align proteins (100 sequences);
- *Fib*: Computes the *n*th Fibonacci number ($n = 50$);
- *Health*: Simulates a national health system (144 cities);
- *NQueens*: Solves the *n*-queens problem ($n = 14$);
- *Sort*: Uses parallel mergesort, transitioning to sequential quicksort and insertion sort (128M integers);
- *SparseLU*: Computes the LU factorization of a sparse matrix (10000×10000 matrix, 100×100 blocks)
- *Strassen*: Multiplies dense matrices (8192×8192 matrix)

We execute with each thread pinned to one core of a 32-core Intel Nehalem-EX shared memory system. Figure 1 shows the speedup of the BOTS applications, which ranges from near-linear to poor. Figure 2 divides the time in 32-thread executions into four categories: sequential equivalent time, work time inflation, overhead time, and idle time. In efficient executions, such as *Alignment*, the sum of the total time across all threads is close to the sequential equivalent time. Overhead, idle time, and work time inflation contribute significantly to the total time of less efficient executions. However, overhead and idle time contribute less than 10% to the total time (except for *Strassen*). Our experiments use cutoffs designed to control the granularity of the tasks for adequate load balance and limited overhead [4]. Nonetheless, all applications except *Alignment* have non-negligible work time inflation.

III. Work Time Inflation and the Impact of NUMA

Work time inflation has hardware and software factors. Software factors can occur in the application (algorithms and their implementation), the compiler (optimizations), and the run time system (task scheduling). Hardware factors include instruction scheduling and memory-related issues (caches, on-chip and off-chip latencies, contention).

Compiler-related issues cause work time inflation (almost half of the overall time) in *SparseLU*. Using ICC with the -ipo flag (interprocedural optimization) improves sequential performance nearly 3X. It also improves parallel performance, but by only 60%, so the optimizations do not scale linearly. This difference in work time inflation is evident even in single-threaded OpenMP executions.

Increasing memory system demands lead to work time inflation in other benchmarks. Non-uniform memory access times reflect physical reality: some memory devices are closer to some processors. Besides physical latency differences, memory device and processor to memory bandwidth constrain the number of sustained concurrent references [9]. Coherence maintenance also incurs additional overhead and latency. Acar et al. [10] characterize cache misses that follow load balancing operations. These cache misses contribute to work time inflation, an important effect even in some programs with small working sets, e.g., *Fib* and *NQueens*.

Strassen uses an algorithm that has lower asymptotic time complexity ($O(N^{2.807})$) than standard matrix multiplication ($O(N^3)$). It recursively decomposes the matrices into submatrices and only uses seven multiplication operations on the submatrices for each submatrix result (compared to eight). However, Strassen's algorithm reduces locality, which leads to less effective cache use and high-latency remote memory accesses. Proposed techniques to improve locality include using to standard matrix multiplication at the base of the recursion and using the more cache-friendly Morton ordering data layout, which also incurs some costs [11].

Sort permutes the elements of a vector, which inherently leads to remote accesses. Tasks compare values in two sublists of the vector, which lacks computational intensity. Efficient execution requires low memory access times as the thread count increases. Figure 3 shows the memory bandwidth available per thread as a function of the number of threads that generate memory reference streams on a 32-core Intel Nehalem machine [9]. Bandwidth per thread is nearly constant for up to 12 threads but decreases with more threads. Ensuring adequate

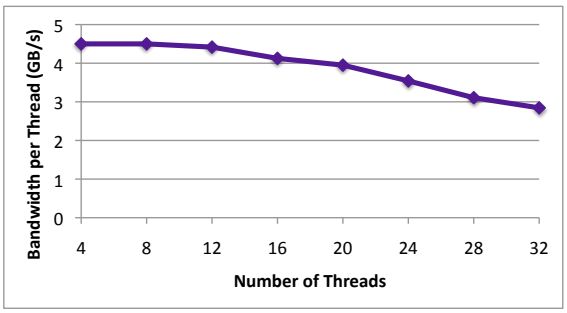


Fig. 3: Bandwidth per thread on the 32-core Intel system.

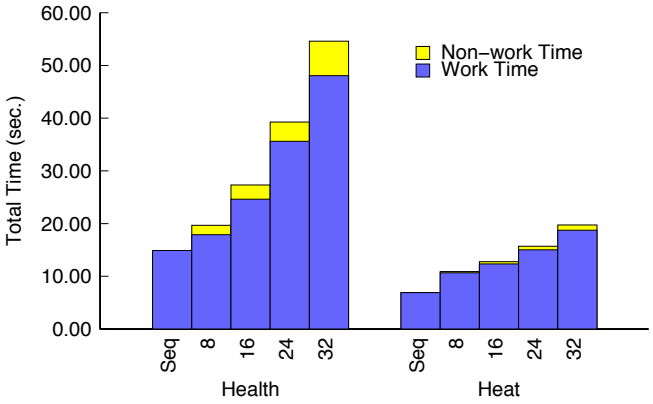


Fig. 4: Execution time break-down (total time over all threads).

memory concurrency for bandwidth-limited applications like *Sort* is critical, especially with increased cores per chip.

Health runs at only 27% efficiency on 32 threads. This time-dependent simulation uses a divide-and-conquer approach to simulate disease-related events (infected population, patients, hospitals) in small villages and to propagate the effects across geographic areas over time. The geographic areas are organized hierarchically so the communication is often localized. In fact, only 2% of patients are transferred between hospitals in different regions. Thus the algorithmic design of *Health* can exploit locality. Poor performance arises from scheduling tasks in a locality-oblivious way. Remote memory accesses and coherence misses occur more often than necessary, a pathology that is magnified with increasing thread counts.

The left bars of Figure 4 sum the time across all threads for parallel executions of *Health*. This time would be constant with perfect scaling and equal to the sequential equivalent time, as the number of threads increases. Instead, work time inflation increases substantially as we use more threads despite using the same input problem size (i.e., strong scaling).

The right bars of Figure 4 show the execution time break-down for *Heat*, a two-dimensional heat diffusion simulation from the Cilk [12] example set that we ported to OpenMP. The simulation uses five-point stencil computations and, like *Health*, generates tasks in a divide-and-conquer decomposition with a cutoff threshold for granularity control. Similarly to

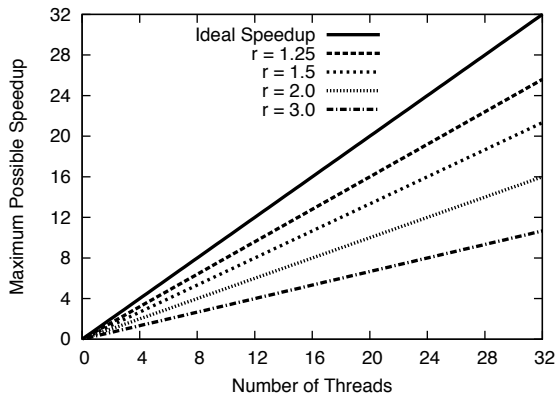


Fig. 5: Speedup limitations due to work time inflation.

Health, *Heat* achieves speedup (11.2X) and parallel efficiency (35%) on a 32 thread execution that are well short of ideal. Work time inflation even contributes 29% of the total time in a single-threaded execution using OpenMP. This inflation is an artifact of parallel code limiting the efficacy of compiler optimization, as seen in *SparseLU*. The remaining work time inflation, which increases as we use more threads, is again due to NUMA effects and memory performance.

We collect hardware performance counter measurements during a sequential execution and a 32-thread parallel execution of the 2D heat simulation. Both read 13.6GB from memory. However, the parallel execution generates 31GB of interconnect traffic between the processor chips, over 400 times the traffic of the sequential execution. This interconnect traffic represents remote loads and cache invalidations that cause work time inflation. Our quad-socket Intel platform, which uses the Quick Path Interconnect (QPI), exhibits significant Non-Uniform Memory Access (NUMA) times. Systems from AMD and IBM exhibit similar latency differences to local and remote memory [13], [14].

Work time inflation limits application speedup. Let r be the ratio of the total work time of a p-processor parallel execution to the sequential work time. The lowest potential execution time is r/p and the highest potential parallel speedup is p/r. With 32 threads, *Health* yields $r > 3$ and *Heat* yields $r > 2.5$. Figure 5 shows the maximum potential speedup for various levels of work time inflation ($r = 1.25$ to 3.0). Even with a relatively small NUMA impact of $r = 1.25$, the maximum speedup is less than 26 with 32 threads. Non-zero overhead and idle times can lead to even lower speedup.

IV. First Touch and Scheduling

OpenMP offers no inherent means to express locality for data to threads or tasks. Thus, performance-oriented users must rely on non-portable solutions such as operating system tools (e.g., libnuma in Linux), third party libraries (e.g., hwloc [15]), or heuristics. Integrated OpenMP locality support would overcome the inconvenience of these methods. However, a common programming idiom in NUMA systems is the first-touch page placement policy [16], [17]. We now examine

```
#pragma omp parallel
{
  #pragma omp for schedule(static)
    for (i = 0; i < n; i++)
      init(data[i]);

  for (step = 0; step < nsteps; step++)
    #pragma omp for schedule(static)
      for (i = 0; i < n; i++)
        compute(data[i]);
}
```

Fig. 6: Simple first-touch initialization under OpenMP.

how this heuristic is used with parallel loops and how current task schedulers fail to support an equivalent idiom for task parallel programs, a role that our locality framework fills.

This idiom assumes an identical schedule of loop iterations in the initialization and computational loops. The first-touch policy places a memory page in memory attached to the socket of the processor that executes the thread that first accesses the page. If the data use in the computational loop iterations mirrors the data use in the initialization loop iterations and if we bind threads to processors with the `OMP_PROC_BIND` environment variable, memory accesses are local. We show an example of this pattern in Figure 6 in which each thread initializes n/p elements of `data` and computes on those elements during each simulation step, making most data accesses local on NUMA systems with the first-touch policy.

Figure 7 shows the analogous code for an OpenMP task parallel divide-and-conquer program with tree-structured data, which uses barriers to ensure the completion of all tasks during initialization and within each step of the simulation loop. Task parallelism in `recursive_init()` spreads the data across memory banks. However, we have no mechanism (not even a non-portable one) to guarantee that `recursive_compute()` task placement mirrors that of `recursive_init()`. Thus, a corresponding `recursive_compute()` task may be scheduled on a different thread that runs on a different socket, in which case we incur non-local accesses and, thus, work time inflation. The lack of a task scheduling analog is unfortunate, since task parallel computations allow easy expression of key computation patterns, such as octree decompositions, for which a flat 1-D loop distribution is not well suited.

V. A Framework for Locality-Based Scheduling

Our framework for locality-based scheduling builds on the notion of a *locality domain*, which consists of:

- Logically, one or more threads and associated storage;
- Physically, one or more cores and physical memory close to them (e.g., a multicore chip and its directly attached memory) to which the system maps the threads;
- An associated (set of) task queue(s) that are distinct from the queues associated with other locality domains;
- A unique integer identifier, its *locality domain identifier*.

We next describe our OpenMP extensions to support locality domains and a prototype implementation of those extensions.

```
void recursive_init(data_t *data) {
  init(data);
  if (!is_leaf(data)) {
    #pragma omp task
      recursive_init(data->left);
    #pragma omp task
      recursive_init(data->right);
    #pragma omp taskwait
  }
}

void recursive_compute(data_t *data) {
  if (!is_leaf(data)) {
    #pragma omp task
      recursive_compute(data->left);
    #pragma omp task
      recursive_compute(data->right);
    #pragma omp taskwait
  }
  compute(data, data->left, data->right);
}

#pragma omp parallel
{
  #pragma omp single
    #pragma omp task
      recursive_init(top);
  #pragma omp barrier

  for (step = 0; step < nsteps; step++) {
    #pragma omp single
      #pragma omp task
        recursive_compute(top);
    #pragma omp barrier
   }
}
```

Fig. 7: Analogous initialization for OpenMP tasks.

A. A Concise API for Programmer-Specified Scheduling

We add the following API calls to OpenMP to allow the programmer to access information about locality domains and to specify placement of tasks on them.

- `omp_child_task_affinity(`*`locality_domain_id`*`)` sets an internal control variable (ICV) that indicates the locality domain on which the run time should place tasks that the currently executing task generates. The *`locality_domain_id`* identifier specifies the locality domain. This call overrides default placement of child tasks, which is the parent task's locality domain.
- `omp_get_num_locality_domains()` returns the total number of locality domains in the system.
- `omp_get_locality_domain_num()` returns the locality domain identifier on which the task is executing.

Figure 8 shows how these calls can distribute tasks and data according to a predictable locality-based schedule. We omit `recursive_compute()`, which is similar to `recursive_init()`. If the run time presents four locality domains, each iteration places the top task on locality domain 0, and the first split places tasks on locality domains 0 and 2.

```
void recursive_init(data_t *data, int nsplits)
{
  init(data);
  if (!is_leaf(data)) {
    #pragma omp task
      recursive_init(data->left, nsplits-1));
    if(nsplits > 0) {
      int dom, nextDom;
      dom = omp_get_locality_domain_num();
      nextDom = dom + (int)pow(2, nsplits-1);
      omp_child_task_affinity(nextDom);
    }
    #pragma omp task
      recursive_init(data->right, nsplits-1);
    #pragma omp taskwait
  }
}

int ndomains, nsplitsTotal;
ndomains = omp_get_num_locality_domains();
nsplitsTotal = (int)log2(ndomains);

#pragma omp parallel
{
  #pragma omp single
  {
    omp_child_task_affinity(0);
    #pragma omp task
      recursive_init(top, nsplitsTotal);
  }
  #pragma omp barrier

  for (step = 0; step < nsteps; step++) {
    #pragma omp single
    {
      omp_child_task_affinity(0);
      #pragma omp task
        recursive_compute(top, nsplitsTotal);
    }
    #pragma omp barrier
  }
}
```

Fig. 8: Code that uses our OpenMP task affinity mechanism.

Secondary splits place tasks on locality domains 0, 1, 2, and 3. We place all other tasks on the locality domain of the task that generates them. Thus, a thread in the same domain during initialization and during each simulation time step executes the subtree of tasks that each second split task generates. Our run time calls can also schedule more irregular and dynamic task layouts, such as those generated by adaptive methods.

B. Run Time Scheduling Policy and Implementation

The OpenMP specification [18] places few restrictions on task scheduling, allowing flexibility in scheduler design and implementation. Our extensions naturally suit a scheduler that has a logical queue per locality domain. Our implementation uses separate queues for each locality domain. An alternative scheme could use a central queue for all locality domains and then dispatch each task to the appropriate locality but is unlikely to scale well.

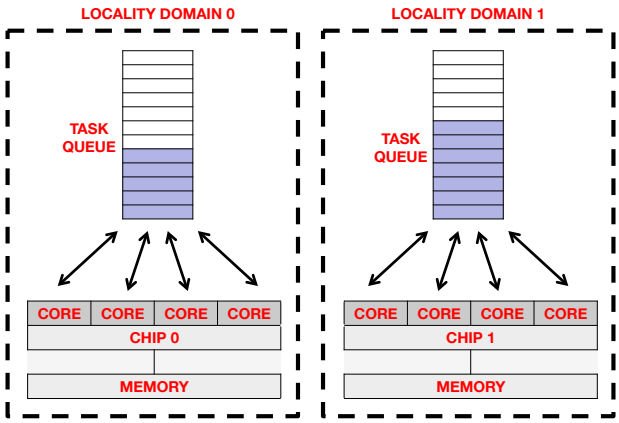


Fig. 9: Mapping locality domains to a two-socket system.

We need a scheduling policy between and within locality domains. `omp_child_task_affinity()` specifies initial task placement, but load balancing could migrate the task to another locality domain. Such migrations may lead to non-local data accesses and, thus, work time inflation. Thus, we provide a *strict* mode that disallows migrations between locality domains but allows load balancing within each locality domain.

We extend the Qthreads hierarchical task scheduler [19]. We use the OpenMP 3.0 support in ROSE [20] to parse OpenMP application code, to perform appropriate semantic analysis, and to generate outlined functions that are mapped to their OpenMP extensions in the Qthreads library [21]. We compile the transformed code with the GNU C/C++ compiler and execute the compiled program with the Qthreads library.

Qthreads determines the system's hardware topology through one of several portable or architecture-specific libraries. We use hwloc [15], which supports x86 NUMA machines well. In the Qthreads hierarchical scheduler, all threads that run on cores on the same chip share a task queue scheduled in a LIFO discipline, which promotes constructive L3 cache sharing and provides natural load balance among those threads. Work stealing [22] balances load between chips.

We map locality domains to the Qthreads representation of the NUMA topology and use per-socket shared queues, as Figure 9 shows. By default, when a task is generated, it is queued on the shared queue that the thread executing the parent task is using. However, if the parent task's task affinity ICV has been set to another locality domain, we place the new task on that domain's queue. If the *strict* mode is set, we disable work stealing between the task queues. Otherwise, tasks may migrate between queues. The run time calls to return the total number of locality domains and the current locality domain map directly to existing Qthreads functions.

VI. Evaluation

We apply our task locality techniques to *Health* and *Heat*. We add API calls to distribute the top level tasks among the available locality domains, but we do *not* change the number or contents of the tasks in the two applications.

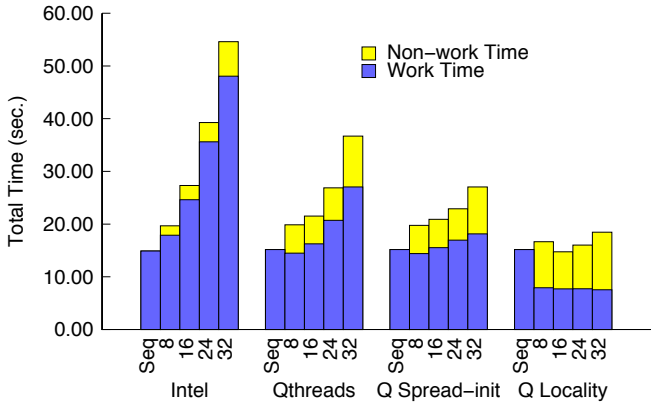


Fig. 10: *Health* execution time summed over all threads.

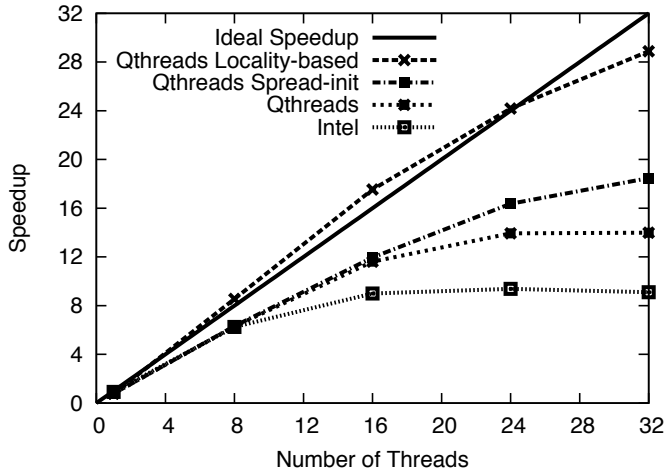


Fig. 11: Speedup on *Health*.

Our test system is a Dell PowerEdge M910 quad-socket blade with four Intel x7550 2.0GHz 8-core Nehalem-EX processors. Each processor has an 18MB shared L3 cache and each core has a private 256KB L2 cache and 32KB L1 data and instruction caches. The blade has 64 dual-rank 2GB DDR3 memory sticks (16 per processor chip) for a total of 128GB. It runs CentOS Linux with a 2.6.35 kernel. We turn off Intel's HyperThreading feature in the BIOS (as is common with HPC environments) and pin one thread to each physical core.

We use the GNU C/C++ 4.4.4 compiler with $-O2$ optimization to obtain sequential times and as the native compiler for the code transformed by ROSE 0.9.5a. We compare to the Intel 11.1 compiler with $-O2$ $-xHost$ $-ipo$ optimization and the Intel OpenMP run time system. The difference in sequential time between the two compilers is negligible on *Health* and more pronounced on *Heat*. For a comparison of parallel executions on absolute terms, we show the elapsed execution time for each application. Timing and speedups represent the best of ten trials for each configuration.

Figure 10 shows the execution time break-down for *Health* using several different configurations. The first set of bars

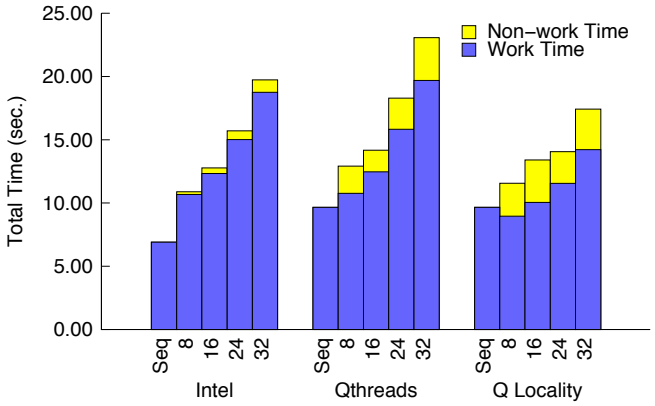


Fig. 12: *Heat* execution time summed over all threads.

Threads	8	16	24	32
Intel	2.46	1.71	1.64	1.69
Q Locality	1.83	0.890	0.646	0.540
% Decrease	25.6	48.0	60.6	68.0

TABLE I: Run times (elapsed) on *Health*.

Threads	8	16	24	32
Intel	1.36	0.799	0.654	0.617
Q Locality	1.30	0.763	0.566	0.520
% Decrease	4.4	4.5	13.5	15.7

TABLE II: Run times (elapsed) on *Heat*.

shows results for executions that use the locality-oblivious Intel run time, as we showed previously in Figure 4. The second set, labeled *Qthreads*, shows results obtained using our base locality-oblivious hierarchical scheduler [19]. The sequential times for Intel and GCC are close (14.9 and 15.2 seconds). The timing results for the parallel executions show more non-work time spent, i.e., combined idle and overhead time, compared to the Intel run time. However, work time inflation is much lower for *Qthreads* (12 seconds), due to the hierarchical scheduler, than for *Intel* (33 seconds). The third set, *Q Spread-init*, shows that we achieve further improvement by using our framework to spread the initialization of the program data across locality domains but without the use of locality-based scheduling in the simulation. This change further reduces work time inflation through better use of memory concurrency. The fourth set, *Q Locality*, uses the spread initialization and also locality-based scheduling in the *strict* mode to ensure that each task executes on a thread in the locality domain to which the data that the task uses is associated. The full and contention-free use of all available cache and memory bandwidth without undue L3 cache invalidations results in work time equivalent to roughly half of the sequential time. The tradeoff is increased non-work time due to a lack of load balancing between locality domains.

Figure 10 presents the *sum* of the total times spent by all threads in an execution. Alternatively, Figure 11 shows the parallel speedup achieved, using ICC and GCC sequential

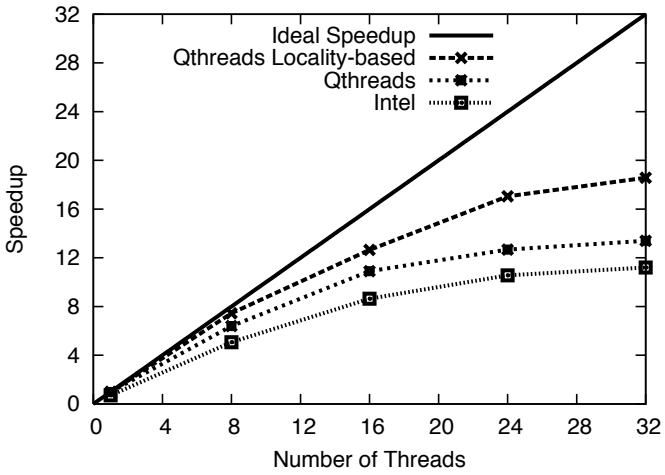


Fig. 13: Speedup on *Heat*.

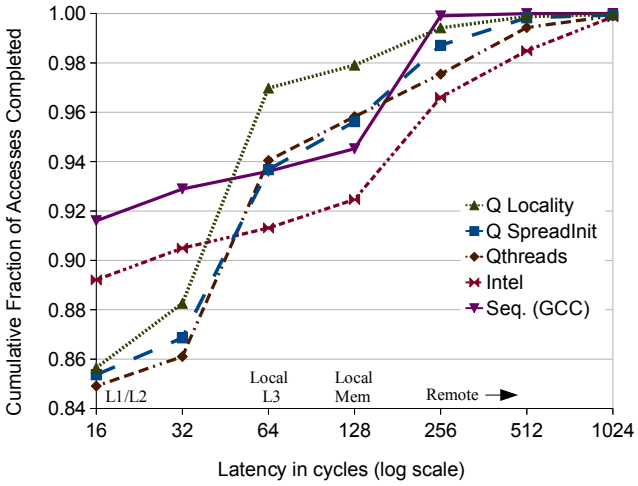


Fig. 14: CDF of *Health* data access latencies.

times for the Intel and Qthreads schedulers, and the *elapsed* times for the parallel executions. *Intel* speedup flattens at 16 threads. *Qthreads* speedup is better, but still flattens at 24 threads. The use of spread initialization in Qthreads results in further improvement, but even it reaches only 18X speedup on 32 threads. Only the Qthreads locality-based scheduling achieves near-linear speedup, even exhibiting slightly super-linear speedup at 16 threads. Table I compares the elapsed times for the Intel and locality-based schedulers.

Figure 12 shows a time breakdown for *Heat*. ICC sequential execution (6.9 sec.) is significantly faster than GCC (9.7 sec.). Despite this gap, *Intel* and *Qthreads* work times for parallel executions that use the same number of threads are close. Work time inflation on 32 threads is near 100% using Qthreads and 170% using Intel's run time. Again, *Qthreads* non-work time is higher. *Heat* uses parallel initialization to spread the data, so the vehicle for further improved performance must be locality-based scheduling during the simulation so tasks execute near the data that they access. We observe from the *Q Locality* results that this scheduling eliminates some but not all work time inflation, with similar non-work times. On 32 threads, it reduces work time inflation by more than half compared to the locality-oblivious scheduler. Since *Heat* uses stencils that require exchanges of data at the boundaries of the grid, some non-local data accesses are inevitable. Thus, even with locality-based scheduling some work time inflation remains. Despite the difference in sequential times using GCC and ICC, the total execution times using locality-based scheduling are lower across the board, as Table II shows. The speedup improvements (Figure 13) are significant although none of the schedulers achieve ideal speedup.

A. Detailed Performance Measurement

To obtain a more complete understanding of run time behavior, we measure performance characteristics of executions that use locality-oblivious and locality-based scheduling. Since we posit that memory latency incurred on non-local accesses causes the observed work time inflation, we measure load latency during the execution of our two examples with Intel's Precise Event Based Sampling (PEBS). Figure 14 plots a cumulative distribution function (CDF) for latency with *Health* using each scheduler with 32 threads, along with results from a sequential execution. For a point (x, y) on the graph, y is the cumulative fraction of loads that have completed in x cycles or less. We observe that for all schedulers, at least 85% of loads complete in 16 cycles or less. Microbenchmarking studies of the Nehalem cache hierarchy and memory subsystem [23] indicate that latencies in this range are hits in the core's L1 and L2 caches. As we proceed across the x-axis, we note the regimes in which accesses represent on-chip L3 cache hits (and hits in L2 caches of cores on the same chip), accesses to memory on the same socket, and finally, accesses to remote L3 caches and memory on other sockets.

Starting from the graph's left, the first difference between the results for the different schedulers is that the sequential execution best uses the per-core L1 and L2 caches, which we expect since multithreaded executions introduce coherence misses. Also, threads on the same chip in the Qthreads schedulers share a queue, and shared queues are cache-friendly for shared caches, e.g., the Nehalem L3, but not individual caches, e.g., L1 and L2. Although we cannot examine its run time source code, we suspect that *Intel* uses per-core work stealing, which is cache-friendly for individual caches. The benefit of the per-chip shared queues to exploit the shared L3 caches manifests in the sharp increase in accesses completed by 64 cycles using Qthreads. The locality-aware scheduler achieves more cumulative completions in this range than even the sequential execution, as it effectively exploits the full L3 cache of the machine with minimal coherence misses between sockets. This scheduler also outperforms *Intel* and sequential executions in the range of local memory accesses by engaging the full memory bandwidth of the machine with minimal accesses to memory on remote sockets. The data points at

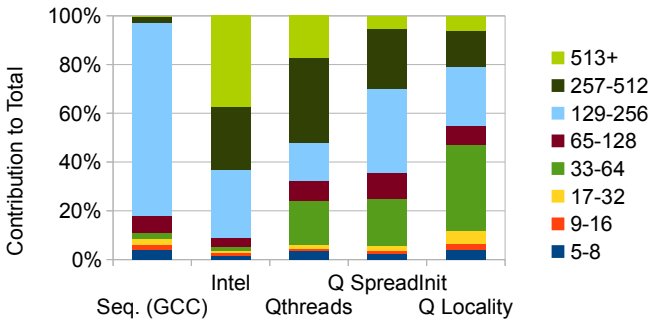


Fig. 15: *Health* percent time spent on various access latencies.

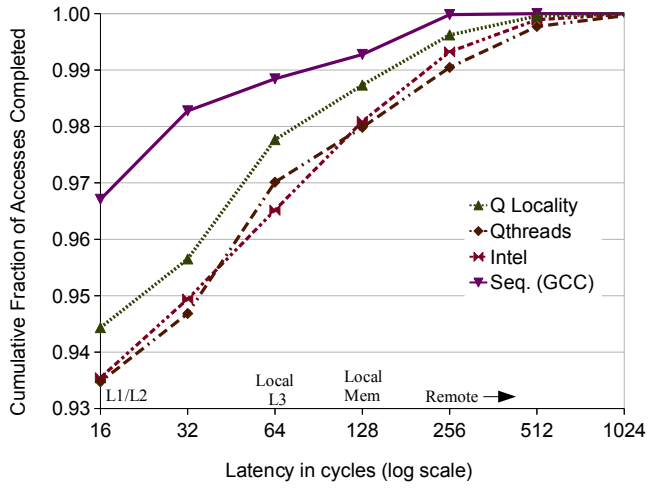


Fig. 16: CDF of *Heat* data access latencies.

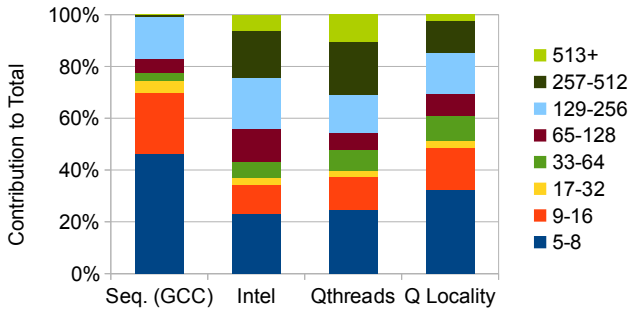


Fig. 17: *Heat* percent time spent on various access latencies.

256 cycles and beyond represent remote memory accesses. The sequential execution, since it only uses a single core on a single socket, shows a negligible number of remote memory accesses that can be attributed to I/O or system processes. The completion rates of the Qthreads and Intel schedulers line up in the same order as the speedup graphs, led by *Q Locality*.

The relative impact of accesses in each range is a function of not only the number of accesses in the range, but also the number of cycles per access in that range. For example, one access that requires 256 cycles to complete contributes the same number of cycles to the total as 16 accesses of 16 cycles. Figure 15 shows the relative contribution of the different ranges of access times, using a conservative estimate: number of accesses *times* the low end of the range, e.g., assume all accesses in the range of 33-64 cycles take 33 cycles. The vast majority of access time in the sequential execution is spent on loads in the local memory regime (129-256). For the locality-oblivious schedulers, accesses of greater than 256 cycles in duration account for almost half of the total *Qthreads* latency and 60% with *Intel*. In addition to low contributions to total latency from high latency loads, locality-based scheduling also shows a much higher contribution of L3 regime loads (about 40%) than any of the other schedulers, and again improves upon the sequential execution in that range.

	Seq. (gcc)	Intel	Qthreads	Q Locality
Health	0.067	34	26	1.0
Heat	0.077	31	31	0.34

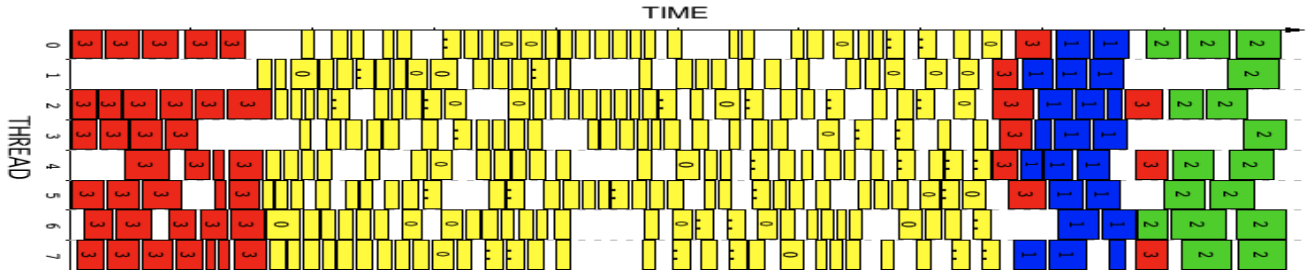
TABLE III: Data Transferred (GB) over QPI between sockets.

Figure 16 shows a CDF of load latency sampled during *Heat* executions. The sequential execution experiences the lowest latency in all regimes. While the latencies incurred with locality-based scheduling are much higher, they are still significantly lower than those incurred with the locality-oblivious schedulers (*Qthreads* and *Intel*). This observation is consistent with our speedup results: locality-based scheduling outperforms locality-oblivious scheduling but falls short of ideal speedup. Figure 17 shows the relative contribution of the different latency ranges. A marked difference between *Health* and *Heat* is evident just from comparing the sequential results: Contributions from L1 and L2 accesses are less than 10% in executions of *Health* but 70% in executions of *Heat*. *Q Locality* sees a lower contribution of high latency loads than the locality-oblivious schedulers.

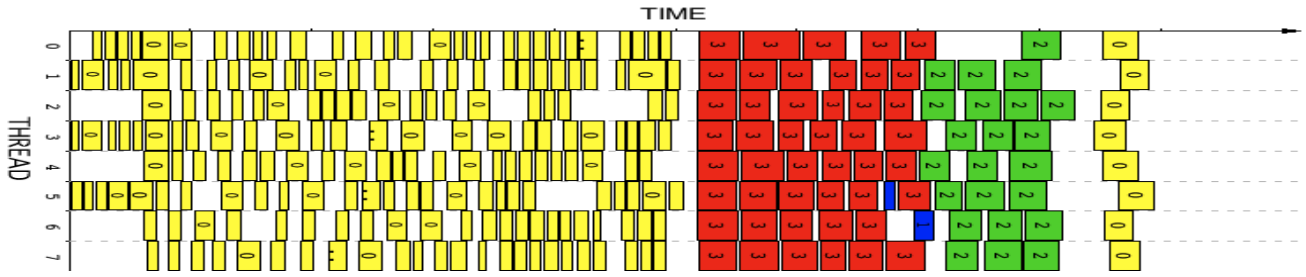
Another interesting metric is QPI traffic, i.e., the amount of data transferred between sockets during execution. Table III shows the measured volumes in gigabytes. The interconnect is quiet during sequential execution, as we might expect. The significant observation is the order of magnitude reduction in QPI traffic from the locality-oblivious schedulers of Intel and Qthreads to the locality-based Qthreads scheduler.

B. Visualizing Observed Task Schedules

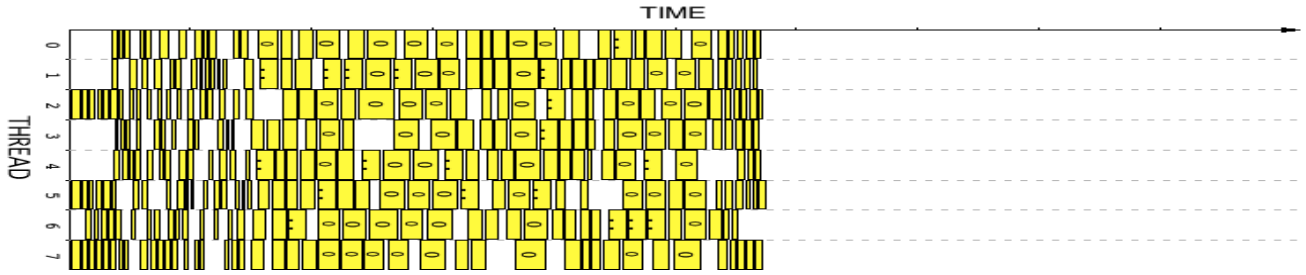
Instrumentation to log actual schedules gives a micro-level view of scheduler behavior to allow a detailed evaluation of the impact of scheduler decisions. We use the Jedule tool [24] to generate visualizations of task schedules, as Figure 18 shows. The three recorded partial schedules are taken from 32 core executions of the health simulation using three different schedulers, with a uniform time axis spanning about 3.1 milliseconds. In each execution, we record the start and end times of each leaf task in the 100th iteration of the simulation loop, the thread number on which the task executes and the locality domain in which the data it uses is located. The vertical axis represents the threads (for the first eight of 32



(a) Locality-oblivious scheduling



(b) Non-strict locality-based scheduling



(c) Strict locality-based scheduling

Fig. 18: Observed schedules of tasks over time during *Health* execution (time step 100, first 8 of 32 threads).

threads) and the horizontal axis represents time. The threads shown (threads 0-7) are pinned to socket 0. Each box is a task. The numbers and colors on each task indicate the number of the socket that is directly linked to the memory bank that contains the data that task uses: yellow, blue, green, and red for sockets 0, 1, 2, and 3. Thus, the tasks numbered 0 and colored yellow use data that resides in memory attached to socket 0 and perform only local memory and cache accesses when executed on threads 0-7. The tasks with other numbers and colors use data local to the other sockets (not shown in the diagram) and the other threads (8-15, 16-23, and 24-31).

Figure 18a illustrates locality-oblivious scheduling. In the early part of the execution, the threads in this locality domain work on tasks that access data that is local to locality domain 3. Later in the execution, tasks that access local data are stolen. These tasks generally execute more quickly, as indicated by the smaller boxes. Eventually, the threads steal more tasks that access data in locality domains 1, 2, and 3.

Figure 18b shows a schedule from locality-based scheduling in *non-strict* mode, i.e., with work stealing allowed between locality domains. This schedule completes in slightly less time than the locality-oblivious schedule. Initially, the threads execute tasks that access data local to the locality domain. Just more than half way through, a thread on another locality domain steals the remaining work, and in turn thread 4 steals tasks from locality domains 3 and 2, and a couple from 1. This behavior occurs due to instantaneous work imbalances that occur even in computations that are on the whole balanced, especially in the later stages of execution. Near the end of the computation, thread 3 reacquires some tasks that access local data, but they require more time now, because their data has been cached in the L3 caches in the other locality domains and must now be invalidated.

Strict locality-based scheduling results in much faster execution, as Figure 18c shows. Threads 0-7 only execute tasks that access local data. When a thread goes idle and no tasks are on the queue, the thread is forbidden to steal from other locality domains and must wait for one of the other threads in its own locality domain to generate more work. The short tasks near the beginning and end of the execution indicate

another benefit of executing only local tasks: better locality between the simulation steps. Local caches are not polluted by non-local data, and, conversely, local data does not end up in remote caches. In contrast, with locality-oblivious scheduling, and even *non-strict* locality-based scheduling, each simulation step can begin with a significant amount of interconnect cross-traffic and cache invalidations.

VII. Related Work

We build on the work of Tallent and Mellor-Crummey [6] that attributes performance loss in task parallel programs to categories. In their work, the sources are (parallelization) overheads and load imbalance; non-local data accesses lead to the decrease in performance that we target. McCurdy and Vetter [25] describe effective strategies to diagnose NUMA issues in the general class of shared memory parallel programs using hardware performance counter measurements. Terboven et al. [26] study the performance of OpenMP task parallel applications on NUMA machines and the impact of task generation patterns using the Intel, Oracle, and GNU OpenMP run time systems.

Seminal work in scheduling of task parallelism focused on the exploitation of cache locality without explicit programmer annotations. The Cilk [12] run time exploits individual per-processor caches when using work stealing [22], while the near-serial-order PDF scheduler of Blelloch et al. [27] exploits shared caches better [28]. Acar et al. [10] derive bounds on cache misses based on the number of steals in programs scheduled using work stealing. Chapel [29] and X10 [30] target clusters comprised of interconnected multicore nodes, and they incorporate notions of locality (*locales* and *places*, respectively) into the languages, primarily to distinguish on-node versus off-node data and computation. In X10, computations on objects that reside in remote places are launched as *asynchronous activities* in those places. The SLAW scheduler [31] demonstrates the possible use of "places" within a cluster node in Habanero Java (an offshoot of X10) and Hierarchical Place Trees [32] map multiple levels of *places* to levels of the cache hierarchy. However, their empirical evaluations focus on loop-level parallelism, not the divide-and-conquer task parallel programs that we target. Similarly, Threading Building Blocks (TBB) includes an affinity scheduler [33] for loops but does not consider any such mechanism for task parallelism. Pilla et al. propose NUMA-aware load balancing for the Charm++ run time system [34].

Proposals in many directions provide locality support for OpenMP. Schmidl et al. create a *distance matrix* [35] to represent distances between cores and use topology-aware thread binding [36] of parallel loop nests. Maranthe and Mueller automate profiling to guide page placement [37]. ForestGOMP [38] coschedules threads that share data on cores on the same socket and migrates pages with migrating threads. They schedule nested loop parallelism but not task parallelism. Huang et al. [39] propose locality-based scheduling in their OpenUH compiler and run time based on new OpenMP directives and clauses for data, loops, and tasks. Their evaluation uses benchmarks based on parallel loops and demonstrates the importance of data distribution on NUMA machines. We have implemented our framework using run time API calls that could be converted to compiler directives. Most importantly, we evaluate and analyze performance using locality-oblivious and locality-based scheduling for divide-and-conquer *task parallel* applications.

VIII. Conclusions and Future Work

We demonstrate that lost efficiency in task parallel applications can be attributed to overhead costs, thread idleness, and work time inflation. The sources of work time inflation include issues in compiler optimization, algorithmic design, the memory hierarchy, and task scheduling.

For some applications, our locality-based task scheduling framework reduces work time inflation attributable to non-local data accesses. The example applications to which we apply the framework in our evaluation are time-dependent simulations that have significant data reuse across iterations of the outer simulation loop. For both applications, global dynamic load imbalance is a much smaller contributor to performance loss than work time inflation, enabling effective use of the *strict* scheduling mode. In the *Health* simulation, data accesses are almost completely confined to the bounds of the locality domain, which enables near-linear speedup using locality-based scheduling. In the *Heat* simulation, some data must be shared between the locality domains when the stencil is applied to points near the boundary of the subspaces assigned to two locality domains. Accordingly, we observe a smaller performance improvement on *Heat* than on *Health*.

Our Locality-based framework does not improve performance of all task parallel applications. The BOTS *Sort* benchmark is memory-bandwidth bound and performs unavoidable data exchanges on the whole input. For such applications, essentially no scheduling strategy can improve performance beyond the limits of memory concurrency on the target machine [9]. Our future work will expand the reach of locality-based scheduling to applications that have a changing and unbalanced locus of work that requires global dynamic load balancing, such as the adaptive Fast Multipole Algorithm (FMA) for N-body simulation, for which our *strict* scheduling mode would be suboptimal.

Acknowledgment

The authors thank the anonymous reviewers for their helpful feedback and RENCI for the use of the Intel Nehalem system in our evaluation. This article has been co-authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for U.S. Government purposes (LLNL-CONF-555492).

REFERENCES

[1] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, pp. 404–418, March 2009.

[2] J. Reinders, *Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism*. Sebastopol, CA: O'Reilly, 2007.

[3] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," *SIGPLAN Notices: OOPSLA '09: 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, vol. 44, no. 10, pp. 227–242, 2009.

[4] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *ICPP '09: Proc. 38th Intl. Conference on Parallel Processing*. IEEE, Sept. 2009, pp. 124–131.

[5] S. L. Olivier and J. F. Prins, "Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 341–360, 2010.

[6] N. R. Tallent and J. M. Mellor-Crummey, "Effective performance measurement and analysis of multithreaded applications," in *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 229–240.

[7] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[8] A. Duran and X. Teruel, "Barcelona OpenMP Tasks Suite," http://nanos.ac.upc.edu/projects/bots, 2010.

[9] A. Mandal, R. Fowler, and A. Porterfield, "Modeling memory concurrency for multi-socket multi-core systems," in *ISPASS 2010: Proc. IEEE Intl. Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, March 2010, pp. 66 – 75.

[10] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *SPAA '00: Proc. 12th ACM Symposium on Parallel Algorithms and Architectures*. ACM, 2000, pp. 1–12.

[11] M. Thottethodi, S. Chatterjee, and A. R. Lebeck, "Tuning Strassen's matrix multiplication for memory efficiency," in *SC98: Proc. 1998 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–14.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI '98: Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1998, pp. 212–223.

[13] M. Funk and R. Peterson, "Of NUMA on POWER7 in IBM i," *IBM Performance Management Resource Library*, http://www.ibm.com/systems/resources/pwrsysperf%5FP7NUMA.pdf, January 2010.

[14] AMD Inc., "Performance guidelines for AMD Athlon(TM) 64 and AMD Opteron(TM) ccNUMA multiprocessor systems," http://support.amd.com/us/Processor%5FTechDocs/40555.pdf, June 2006.

[15] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *PDP 2010: Proc. 18th Euromicro Intl. Conference on Parallel, Distributed and Network-Based Processing*. Pisa, Italia: IEEE Computer Society, Feb. 2010, pp. 180–186.

[16] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, "NUMA policies and their relation to memory architecture," in *ASPLOS '91: Proc. 4th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1991, pp. 212–221.

[17] D. S. Nikolopoulos, E. Artiaga, E. Ayguadé, and J. Labarta, "Exploiting memory affinity in OpenMP through schedule reuse," *SIGARCH Computer Architecture News*, vol. 29, no. 5, pp. 49–55, Dec. 2001.

[18] OpenMP Architecture Review Board, "OpenMP API, Version 3.0," May 2008.

[19] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *ROSS '11: Proc. Intl. Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2011, pp. 49–56.

[20] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries," in *IWOMP 2010: Proc. 6th Intl. Workshop on OpenMP*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds., vol. 6132. Springer, 2010, pp. 15–28.

[21] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *IPDPS 2008: Proc. 22nd IEEE Intl. Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

[22] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," in *SFCS '94: Proc. 35th Annual Symposium on Foundations of Computer Science*. IEEE, Nov. 1994, pp. 356–368.

[23] D. Molka, D. Hackenberg, R. Schone, and M. Muller, "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system," in *PACT '09: Proc. 18th Intl. Conference on Parallel Architectures and Compilation Techniques*, Sept. 2009, pp. 261 –270.

[24] S. Hunold, R. Hoffmann, and F. Suter, "Jedule: A tool for visualizing schedules of parallel applications," in *ICPP 2010: Proc. 39th Intl. Conference on Parallel Processing Workshops*. IEEE Computer Society, 2010, pp. 169–178.

[25] C. McCurdy and J. S. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *ISPASS 2010: IEEE Intl. Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, March 2010, pp. 87–96.

[26] C. Terboven, D. Schmidl, T. Cramer, and D. an Mey, "Assessing OpenMP tasking implementations on NUMA architectures," in *IWOMP 2012: Proc. 8th Intl. Workshop on OpenMP*, ser. Lecture Notes in Computer Science, B. M. Chapman, F. Massaioli, M. S. Müller, and M. Rorro, Eds., vol. 7312. Springer, 2012, pp. 182–195.

[27] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM*, vol. 46, no. 2, pp. 281–321, 1999.

[28] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, "Scheduling threads for constructive cache sharing on CMPs," in *SPAA '07: Proc. 19th ACM Symposium on Parallel Algorithms and Architectures*. ACM, 2007, pp. 105–115.

[29] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Intl. Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[30] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proc. 20th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*. ACM, 2005, pp. 519–538.

[31] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2010, pp. 341–342.

[32] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *LCPC 2009: 22nd Intl. Workshop on Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, Eds., vol. 5898. Springer, 2010, pp. 172–187.

[33] M. Voss and T. Wilmarth, "Intel Threading Building Blocks: Ready for non-uniform memory access platforms," http://isdlibrary.intel-dispatch.com/vc/2724/treadingbuildingblocks%5F110609.pdf, 2009.

[34] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, A. Bhatele, P. O. A. Navaux, J.-F. Méhaut, and L. V. Kalé, "Improving parallel system performance with a NUMA-aware load balancer," INRIA-Illinois Joint Laboratory on Petascale Computing, Urbana, IL, Tech. Rep. TR-JLPC-11-02., 20011. [Online]. Available: http://hdl.handle.net/2142/25911

[35] D. Schmidl, C. Terboven, and D. an Mey, "Towards NUMA support with distance information," in *IWOMP 2011: Proc. 7th Intl. Workshop on OpenMP*, ser. Lecture Notes in Computer Science, B. M. Chapman, W. D. Gropp, K. Kumaran, and M. S. Müller, Eds., vol. 6665. Springer, 2011, pp. 69–79.

[36] D. Schmidl, C. Terboven, D. an Mey, and H. M. Bücker, "Binding nested OpenMP programs on hierarchical memory architectures," in *IWOMP 2010: Proc. 6th Intl. Workshop on OpenMP*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds., vol. 6132. Springer, 2010, pp. 29–42.

[37] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for ccNUMA systems," in *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006, pp. 90–99.

[38] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the execution of OpenMP applications for multicore architectures," in *IPDPS 2010: Proc. 25th IEEE Intl. Parallel and Distributed Processing Symposium*. IEEE, April 2010, pp. 1–10.

[39] L. Huang, H. Jin, L. Yi, and B. M. Chapman, "Enabling locality-aware computations in OpenMP," *Scientific Programming*, vol. 18, no. 3-4, pp. 169–181, 2010.